Objektorientierte Programmierung

Philipp Brendel

7. April 2004

Ausarbeitung zum Vortrag im Proseminar **Programmiersysteme**
Lehrstuhl Programmiersysteme, Prof. Gert Smolka
Betreuer: Andreas Rossberg

Übersicht

Im ersten Kapitel werden Herkunft, Begrifflichkeiten und Konzepte objektorientierter Programmiersprachen (OOP) vorgestellt. Die folgenden beiden Kapitel stellen zwei bekannte OOP, Java und Smalltalk, vor und gehen jeweils kurz auf Designziele und die Ausprägung der im ersten Kapitel besprochenen Eigenschaften ein. Das letzte Kapitel stellt vorteilhafte Aspekte objektorientierten Designs seinen Schwächen gegenüber.

1 Konzepte der OOP

1.1 Ursprung der OOP

Die erste objektorientierte Programmiersprache war Simula67, entwickelt in den 1960er Jahren von Ole-Johan Dahl und Kristen Nygaard am Norwegian Computing Center. Simula sollte die Programmierung sogenannter *ereignisbasierter Simulationen* erleichtern, die darin bestehen, dass aus der Simulation einer Menge unterschiedlicher Ereignisse neue Ereignisse erzeugt werden, die ihrerseits simuliert werden müssen usw.
Will man diesen Ablauf nun programmieren, indem man eine Schlange von Ereignissen sowie eine Simulationsprozedur implementiert, um die einzelnen Ereignisse auszuwerten, so kann man das auf besonders elegante Weise erreichen, wenn man über

- eine Datenstruktur, die eine Schlange darstellt, die unterschiedliche Ereignisse enthalten kann, sowie
- eine Simulationsprozedur, die unterschiedlichen Code ausführt, je nachdem, welche Art von Ereignis sie jeweils simulieren soll (was Fallunterscheidungen seitens des Programmierers über die verschiedenen Arten von Ereignissen ersetzt)

verfügt. Der erste Punkt erfordert in einer getypten Sprache eine Form von *Subtyping*; der zweite Punkt macht *dynamische Bindung* notwendig. Im nächsten Abschnitt werden diese Eigenschaften und einige andere vorgestellt.

1.2 Grundbegriffe der OOP

Allen objektorientierten Programmiersprachen ist irgendeine Form von *Objekt* gemein. Dabei ist die genaue Definition dieses Begriffs von Sprache zu Sprache verschieden; allgemein handelt es sich jedoch bei einem Objekt um einen Wert, der Daten und die auf diesen Daten definierten Operationen vereint.
Man unterscheidet hierbei verschiedene Gruppen von Sprachen anhand der Art und Weise, wie sie die Erzeugung neuer Objekte handhaben. *Klassenbasierte OOP* erzeugen neue Objekte anhand von Schemata, die die in jedem Objekt enthaltenen Daten sowie die auf Objekten möglichen Operationen definieren. *Prototypenbasierte OOP* erzeugen neue Objekte, indem bereits vorhandene Objekte *geklont* und um neue Daten oder Operationen erweitert werden. Sowohl Java als auch Smalltalk sind klassenbasierte Sprachen.

Über den Objektbegriff hinaus existieren noch andere Konzepte, die in OOP anzutreffen sind:

Dynamische Bindung Die Implementierung eines Objekttyps entscheidet darüber, welcher Code tatsächlich ausgeführt wird, wenn zur Laufzeit eine Prozedur auf das entsprechende Objekt angewendet wird; d.h. die Entscheidung findet dynamisch zu Laufzeit statt, nicht statisch zu Compilezeit. Am Beispiel der ereignisbasierten Simulationen: Unterschiedliche Implementierungen des Typs “Ereignis” werden durch unterschiedliche Implementierungen der Simulationsprozedur behandelt, zwischen denen zur Laufzeit ausgewählt wird.

Subtyping In ungetypten Sprachen ist es trivial, Objekte mit unterschiedlicher Struktur z.B. beim Einfügen in eine Schlange (s.o.) gleich zu behandeln. In Anwesenheit von Typen ist dies jedoch nicht ohne weiteres möglich. Man benötigt also eine Methode um zu definieren, welche Operationen auf welchen Typen zulässig sind. Informell gesagt kann man ein Objekt A durch ein Objekt B ersetzen (z.B. beim Aufruf der Einfügeprozedur einer Schlange), wenn B mindestens die gleiche “Funktionalität” bietet wie A. Später gehe ich noch etwas detaillierter auf Subtyping ein.

Enkapsulierung Die Verwendung von abstrakten Datentypen erfolgt stets durch eine Form von Interface, d.h. Manipulation des ADT ist nur durch zum ADT gehörende Prozeduren möglich, wodurch die Einzelheiten der Implementierung versteckt werden. Das selbe Konzept kommt auch in vielen

OOP zum Einsatz: Ein Objekt kann nur durch seine "eigenen" Prozeduren verändert werden.

Vererbung In klassenbasierten Sprachen ist es möglich, eine neue Klasse durch "Ableitung" von einer bereits bestehenden zu erzeugen. Dadurch *erbt* die neue Klasse Daten und Operationen. Das reduziert den Programmieraufwand, da gleiche Prozeduren wiederverwendet werden können und nur wirklich neue Funktionalität der Klasse explizit implementiert werden muss.

Klonen In prototypenbasierten Sprachen werden neue Objekte erzeugt, indem vorhandene Objekte kopiert (geklont) werden und neue Funktionalität hinzugefügt wird.

1.3 Ein Beispiel in ML

In diesem Abschnitt erläutere ich einige der eben vorgestellten Eigenschaften objektorientierter Programmierung am Beispiel einer Implementierung eines Zählers in ML, indem ich auf einige Aspekte der Implementierung hinweise, die den oben besprochenen Konzepten nahekommen.
Der Zähler soll lediglich zwei Operationen unterstützen: Abfrage und Erhöhung, dargestellt durch Prozeduren `get : unit -> int` und `inc : unit -> unit` . Der Typ der Zählerobjekte ist demnach einfach ein Record, das diese beiden Prozeduren enthält: `counter = { get : unit -> int, inc : unit -> unit }`
Als Möglichkeit zur Objekterzeugung dient die Prozedur `newCounter : unit -> counter` , die mit jedem Aufruf einen neuen Zähler erstellt und initialisiert.

```
type counter = { get:unit -> int, inc:unit -> unit }

fun newCounter () = let
                        val n = ref 0
                    in
                        {
                          get = fn () => !n,
                          inc = fn () => n := !n + 1
                        }
                    end

> val newCounter = fn : unit -> {get : unit -> int,
                                 inc : unit -> unit}
```

Das Beispiel macht Gebrauch von Enkapsulierung: Auf die Integerreferenz `n`, die den Zählerstand enthält, können lediglich die beiden Prozeduren des Records zugreifen. Somit ist die konkrete Implementierung des Zählers vor dem Anwender verborgen, der nur `get` und `inc` verwenden kann, um den Zählerstand zu manipulieren. Anwendungsbeispiel:

```
- val c = newCounter ()
> val c = {get = fn, inc = fn} : {get : unit -> int,
                                  inc : unit -> unit}
- val i = #get(c) ()
> val i = 0 : int
- #inc(c) ()
> val it = () : unit
- val j = #get(c) ()
> val j = 1 : int
```

Der Entwickler des Zählers kann nach Belieben dessen Implementierung durch eine neue austauschen, solange er das vereinbarte Interface einhält, das durch den Typ `counter` gegeben ist.

Man betrachte die Prozedur

```
- fun f (c : counter) = (#inc(k) (); #get(k) ())
> val f = fn : {get : unit -> int, inc : unit -> unit} -> int
```

Werte vom Typ `counter` sind gültige Argumente von `f`. Tatsächlich sind diese Werte in ML auch die einzig gültigen Argumente. Es ist jedoch offensichtlich, dass innerhalb der Prozedur über das Argument nur bekannt ist, dass es sich um ein Record mit zwei Prozeduren eines bestimmten Typs handelt - genau das besagt `counter`. Was hindert `f` also daran, auch Werte zu akzeptieren, die zusätzliche Felder besitzen? Um diesen Gedanken weiterzuverfolgen, nehmen wir an, dass wir in einer hypothetischen ML-Variante arbeiten, die genau dieses Vorgehen erlaubt.
Um das Beispiel auszuführen, wird der Zähler um eine Prozedur erweitert, die es ermöglicht, ihn zurückzusetzen. Der neue Recordtyp namens `reset_counter` besitzt also alle Labels, die auch `counter` umfasst, sowie das zusätzliche Label `reset`, das eine Prozedur vom Typ `unit -> unit` fasst.

```
type reset_counter = {
                         get : unit -> int,
                         inc : unit -> unit,
                       reset : unit -> unit
                     }
```

```
fun newResetCounter () = let
                             val n = ref 0
                         in
                             {
                                 get = fn () => !n,
                                 inc = fn () => n := !n + 1,
                               reset = fn () => n := 0
                             }
                         end

> val newResetCounter = fn :
    unit -> {get : unit -> int, inc : unit -> unit,
             reset : unit -> unit}
```

In unserem idealisierten ML-Dialekt sind nun Werte vom Typ `reset_counter` gültige Argumente für Prozeduren wie `f`, da sie den gleichen "Funktionsumfang" aufweisen; das zusätzliche Feld richtet keinen Schaden an. Anwendungsbeispiel:

```
- val c = newCounter ()
> val c = { get = fn, inc = fn } : counter
- f c
> val it = 1 : int
- val c' = newResetCounter ()
> val c' = { get = fn, inc = fn, reset = fn } : reset_counter
- f c'
> val it = 1 : int
```

Es ist uns nun insbesondere möglich, unterschiedliche Implementierungen des Typs `counter` oder eines erweiterten Typs an `f` zu übergeben. Zum Entstehungszeitpunkt der Prozedur ist daher also nicht bekannt, welcher Code ausgeführt wird, wenn `f` angewendet wird; diese Entscheidung trifft die Implementierung des Zählers.

Das Konzept, Typen als "Spezialisierung" anderer Typen anzusehen, wird als *Subtyping* bezeichnet. Im nächsten Abschnitt formalisiere ich diesen Begriff, da er für getypte OOP äußerst wichtig ist.

1.4 Subtyping

Es wurde behauptet, Subtyping führe eine gewisse "Ersetzbarkeit" von Werten eines Typs durch Werte eines anderen Typs ein. Ein Typ σ ist ein *Subtyp* eines Typs τ, wenn in jedem Kontext, der einen Wert vom Typ τ erwartet, ein Wert vom Typ σ verwendet werden kann. Man schreibt dann $\sigma \leq \tau$. In unserem Beispiel bedeutet das, dass ein Subtyp eines Recordtyps genauso

viele oder mehr Labels definieren darf, nicht aber weniger. Um ein Typsystem mit Typrelation $\vdash \subseteq \mathit{Typ} \times \mathit{Typ}$ um Subtyping zu erweitern, ist zunächst eine *Subsumtionsregel* erforderlich:

$$\frac{\Gamma \vdash t : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash t : \tau}$$

Diese Regel drückt die obige Beschreibung der Austauschbarkeit formal aus. Jetzt können die Regeln der Subtypingrelation folgen, wobei folgende Eigenschaften zu beachten sind:

- *Reflexivität*: $\forall \sigma : \sigma \leq \sigma$
- *Transitivität*: $\forall \sigma, \tau, \rho : \sigma \leq \tau \wedge \tau \leq \rho \Rightarrow \sigma \leq \rho$
- *Antisymmetrie*: $\forall \sigma, \tau : \sigma \leq \tau \wedge \tau \leq \sigma \Rightarrow \sigma = \tau$

Das macht die Subtypingrelation zu einer Teilordnung auf der Menge der Typen. Um nun Subtyping für die im Beispiel verwendeten Recordtypen zu ermöglichen, benötigen wir folgende Regeln (zusätzlich zu den Regeln für die oben genannten Eigenschaften):

$$\{\sigma_1, \ldots, \sigma_{n+1}\} \leq \{\sigma_1, \ldots, \sigma_n\} \qquad \frac{\forall i \in \{1, \ldots, n\} : \sigma_i \leq \tau_i}{\{\sigma_1, \ldots, \sigma_n\} \leq \{\tau_1, \ldots, \tau_n\}}$$

Die erste der beiden Regeln erklärt die schon im Beispiel gesehene Beziehung. Die zweite Regel erlaubt es, Subtyping in der "Tiefe" zu betreiben: Enthält beispielsweise ein Label eines Records wiederum ein Record, so gelten die Subtypingregeln natürlich auch für dieses Teilobjekt, nicht nur für das umschließende Record.
Die Subtypingrelation lässt sich allerdings nicht nur auf Records definieren. Als weiteres Beispiel sei die Regel für Prozedurtypen gegeben:

$$\frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

Wegen der sogenannten *Kontravarianz* im linken Teiltyp ist diese Regel besonders interessant (s.u.); sie wird auch als *Kontravarianzregel* bezeichnet.

2 Java

Dieser Abschnitt stellt die objektorientierte Programmiersprache *Java* vor, ihren Ursprung und die Schwerpunkte ihres Designs.

2.1 Entstehungsgeschichte

Java wurde von einem Entwicklerteam um James Gosling im Auftrag von Sun Microsystems Anfang der 90er Jahre entwickelt. Die Sprache sollte ursprünglich *Oak* heißen und als Programmierumgebung eines Gerätes Verwendung finden, das als "set-top box" bezeichnet wurde. Dabei handelte es sich um ein Peripheriegerät für Fernseher, das es dank einer Netzwerkverbindungsmöglichkeit erlauben sollte, den Fernseher als Webbrowser oder als Anzeigegerät für Graphikanwendungen zu nutzen oder gar mit Fernsehprogrammen zu interagieren. Dabei war es unabdingbar, gewisse Eigenschaften der Sprache zu gewährleisten, um Abstürze des Systems zu vermeiden (da das Gerät letztendlich ohne Überwachung eines Programmierers funktionieren sollte) und das Laden von Programmen über ein Netzwerk zu ermöglichen. Diese Merkmale machten Java zu einer erfolgreichen Sprache für Internetanwendungen.

2.2 Designziele

Java ist eine getypte, klassenbasierte OOP. Sie verfügt über eine Garbage Collection und wird interpretiert (Code läuft auf einer virtuellen Maschine, s.u.). Wichtige Ziele beim Design von Java waren:

Offenheit Die Sprache soll möglichst dynamisch, d.h. zur Laufzeit erweiterbar sein. Unter diesen Begriff fallen beispielsweise

- *Dynamisches Linking*, das es ermöglicht, benötigte Objektdateien genau dann zu laden, wenn sie benötigt werden, so dass nicht alle kompilierten Daten statisch gelinkt werden müssen. Hierbei können die Daten sowohl von Festplatte als auch aus dem Netzwerk geladen werden, was dem ursprünglichen Verwendungszweck von Java zugute kommt und natürlich auch für jede Internetprogrammiersprache wichtig ist.
- *Typim- und -export zur Laufzeit*: Für besondere Flexibilität der Anwendungen sorgt die Möglichkeit, Klassen zur Laufzeit zu (de-)serialisieren, was bedeutet, dass statisch nicht alle zu verwendenden Typen bekannt sein müssen.

Portabilität Java-Programme werden nicht zu Maschinencode, sondern zu sogenanntem *Bytecode* kompiliert, der plattformunabhängig definiert ist. Diese Bytecodedateien werden dann auf einer virtuellen Maschine, der *Java Virtual Machine (JVM)*, ausgeführt. Diese umfasst einen leicht portierbaren Befehlssatz, der für jedes Betriebssystem implementiert werden muss. Ist dies einmal geschehen, können beliebige Java-Programme kompiliert und auf dem entsprechenden System ausgeführt werden.

Sicherheit Die Sprache verfügt über einige Features, die die Sicherheit des Programmiersystems erhöhen, d.h. es erschweren, durch Verwendung bestimmter Programmiertechniken unrechtmäßigen Zugriff darauf zu erlangen. Beispiele für Sicherheitsfeatures sind:

- *Arraygrenzenüberprüfung zur Laufzeit*: In Sprachen, die diese Technik nicht unterstützen, ist es möglich, Eingaben über ein Netzwerk an eine Anwendung zu senden, die zu groß für den Puffer sind, der sie aufnehmen soll. Dadurch kann die Rücksprungadresse der Empfängerprozedur überschrieben werden, was dazu führt, dass unbemerkt Code ausgeführt wird, den der "Angreifer" bestimmt. In Anwesenheit einer Grenzenüberprüfung kann das nicht vorkommen.

- *Codesignaturen*: Kompilierte Java-Programme können mit Wasserzeichen versehen werden, die es jedem Empfänger von Code ermöglichen, den Codeproduzenten zu identifizieren und gegebenenfalls das empfangene Programm abzulehnen.

- *"Sandboxing"*: Die JVM überprüft Code, bevor sie ihn ausführt. Dabei kann es dazu kommen, dass sie entscheidet, einen Befehl eines Java-Programms nicht auszuführen, da nicht alle Aktionen erlaubt sind. In diesem Zusammenhang spielen verschiedene Mechanismen eine Rolle, auf die ich nicht näher eingehen werde:

  - der *Class Loader*
  - der *Bytecode Verfier*
  - der *Security Manager*
  - *JVM Runtime Checks*

2.3 Sprachkonzepte

Wie bereits erwähnt, ist Java klassenbasiert. Eine Klasse definiert die Prozeduren (*Methoden*) sowie die Daten (*Felder* oder *Instanzvariablen*) ihrer Instanzen. Objekte werden durch Verwendung sogenannter *Konstruktoren* erzeugt, die in Verbindung mit dem Schlüsselwort `new` die Klasse instanziieren (siehe die Prozedur `newCounter` im ML-Beispiel).

Vererbung Jedes Objekt ist also eine Instanz einer Klasse; jede Klasse erbt von genau einer anderen Klasse (bis hin zu einer obersten Klasse namens *Object*), deren Code sie wiederverwenden oder spezialisieren, d.h. an die eigenen Bedürfnisse anpassen, kann.

Enkapsulierung Java unterstützt sowohl für die Felder als auch für die Methoden, die eine Klasse definiert, verschiedene Ebenen der Kapselung. Der Programmierer kann für jedes Feld und jede Methode eine von vier *Zugriffbeschränkungen* wählen:

- *public*: jeder darf auf das Element zugreifen
- *private*: nur Instanzen der Klasse dürfen auf das Element zugreifen
- *protected*: Instanzen der Klasse sowie abgeleiteter Klassen dürfen auf das Element zugreifen
- *package*: Klassen desselben *packages* (einer Sammlung von Klassen, ähnlich den *namespaces* in C++) dürfen auf das Element zugreifen

Subtyping Im Gegensatz zu der im Kapitel *Konzepte der OOP* vorgestellten Subtypingrelation, die durch die Struktur der Typen implizit eingeführt wurde (*strukturelles Subtyping*), definiert der Java-Programmierer die Subtypingbeziehung explizit: Sie entspricht direkt der Vererbungshierarchie der Klassen (*nominelles Subtyping*). Eine Instanz einer Subklasse kann also in jedem Kontext verwendet werden, in dem die Verwendung einer Instanz der Oberklasse zulässig ist.

2.4 Codebeispiel

Um ein kleines Beispiel einer Klassendefinition und der Verwendung in beliebigem Code zu geben, folgt nun die Implementierung des Zählers aus dem vorigen Kapitel in Java. Zunächst definieren wir eine Klasse, die den Zähler kapselt:

```
class Counter extends Object {
  private int n;
  public Counter() { n = 0; }
  public int  get() { return n; }
  public void inc() { n = n + 1; }
}
```

Der Konstruktor `Counter()` entspricht, wie bereits erwähnt, der Prozedur `newCounter` im ML-Beispiel. In Java ermöglicht es die Definition der Variablen `n` als *private*, sie vor Zugriff von außen zu schützen. Die restlichen Prozeduren folgen dem gleichen Konzept wie die entsprechenden Prozeduren in ML.
In einem Codeblock kann ein Zähler folgendermaßen verwendet werden:

```
Counter c = new Counter();
int i = c.get();
```

```
c.inc();
int j = c.get();
```

Der Konstruktor wird mit Hilfe von `new` aufgerufen. Am Ende der Berechnung hat `i` den Wert 0 und `j` den Wert 1.

3 Smalltalk

Dieser Abschnitt stellt die Sprache *Smalltalk* vor und nennt, wie im vorigen Kapitel zu Java, Designziele und Sprachkonzepte.

3.1 Entstehungsgeschichte

Smalltalk ist eine der ältesten objektorientierten Programmiersprachen. Die Sprache wurde in den 1970er Jahren im Xerox PARC (Palo Alto Research Center) entwickelt. Sie war ursprünglich dazu gedacht, in einem Projekt namens *DYNABOOK* eingesetzt zu werden: Unter Alan Kays Leitung sollte in den Siebzigern ein tragbarer, personalisierter Computer entstehen, dessen Interface vollständig in Smalltalk implementiert würde. Smalltalk sollte außerdem das zentrale Entwicklungstool für die Software des Dynabook sein. Da dies auch die Benutzung durch nicht besonders eingehend geschulte Programmierer bedeuten würde, war das Design von Smalltalk besonders auf Einfachheit ausgelegt (s.u.). Fragen der Effizienz traten dabei etwas in den Hintergrund - man nahm an, die Hardwareentwicklung würde diese Defizite irgendwann ausgleichen.

3.2 Designziele

Einfaches und elegantes Konzept Das Schlagwort von Smalltalk lautet: “Alles ist ein Objekt” (später hierzu noch ein interessantes Beispiel). Diese extreme Ausdehnung der Objektmetapher ermöglicht es, schnell in die Programmierung einzusteigen, da keine komplizierten Sprachkonstrukte und nur sehr wenig Syntax erlernt werden müssen.

Mühelose Entwicklung Um die Einheitlichkeit der Sprache zu unterstützen, umfassen Implementierungen von Smalltalk für gewöhnlich eine graphische Benutzeroberfläche, die zur Programmierung verwendet wird. Diese erfolgt in Browserfenstern, die per Maus bedient werden - im Gegensatz zur “linearen” Programmierung anderer Sprachen, die im Wesentlichen durch Erstellen von Textdateien geschieht.

Umfangreiche Klassenbibliothek Obwohl die Sprache selbst äußerst simpel ist, lassen sich auch mächtige Anwendungen leicht erstellen, da für alle gängigen Programmieraufgaben bereits vorgefertigte Klassen bestehen.

3.3 Sprachkonzepte

Smalltalk ist eine dynamisch getypte, klassenbasierte OOP. Programme stellt man sich als eine Abfolge von Nachrichten vor, die an Objekte geschickt werden. Die Objekte entscheiden, ob sie die Nachricht erkennen; falls ja, wird ein der Nachricht zugeordneter Code ausgeführt, der *Methode* genannt wird. Genau wie Java kennt die Sprache **Vererbung** und **Enkapsulierung**, die schematisch vorgegeben ist: Die Prozeduren eines Objektes sind öffentlich verfügbar, die Datenfelder sind geschützt, also nur von Instanzen von Subklassen und der Klasse selbst aus zugreifbar.
Von **Subtyping** kann man in einer dynamisch getypten Sprache eigentlich nicht sprechen; statisch kann man natürlich instanzen beliebiger Klassen als Empfänger für alle Nachrichten verwenden. Zur Laufzeit stellt sich dann heraus, ob das Objekt die Nachricht versteht. Ist dies nicht der Fall, so wird eine Ausnahme ausgelöst, sofern das Objekt nicht die Methode `doesNotUnderstand` implementiert, mit der auch unbekannte Nachrichten abgefangen und behandelt werden können.
Das Kernkonzept der Sprache ist, wie schon erwähnt, die Idee, dass alles ein Objekt ist. Das trifft natürlich auf die "gewöhnlichen" Objekte zu, die auch aus anderen Sprachen bekannt sind. Aber da in Smalltalk auch Klassen Objekte sind, ergeben sich interessante Beziehungen: Instanzen einer Klasse werden durch Senden der Nachricht `new` an das Klassenobjekt erzeugt. Wie aber werden Klassen erzeugt? Wiederum durch Senden einer Nachricht, diesmal an ein Objekt einer "höheren Ebene", ein Metaklassenobjekt.

3.4 Codebeispiel

Um auch für Smalltalk ein Anwendungsbeispiel zu geben, folgt nun die Implementierung des Zählers. Die einzelnen Codefragmente für die Klassendefinition sind dabei allerdings aus dem Zusammenhang des graphischen Browsers gerissen; der Leser muss sich den semantischen Rahmen selbst hinzudenken.

```
Object subclass: #Counter
       instanceVariableNames: 'n'
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Example-Classes'

initialize
  n <- 0

get
  ^ n
```

```
inc
  n <- (n + 1)
```

Besonders interessant ist hierbei die Definition der Klasse `Counter`, die durch Senden der Nachricht
`subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:`

an das Klassenobjekt `Counter` geschieht. Die Klasse kann in einem Codeblock folgendermaßen verwendet werden:

```
c <- Counter new.
i <- c get.
c inc.
j <- c get.
```

Am Ende des Blocks hat `i` den Wert 0, `j` den Wert 1.

4 Diskussion

Im ersten Abschnitt wurden einige Konzepte objektorientierter Programmierung vorgestellt. Die beiden folgenden Abschnitte zeigten am Beispiel zweier prominenter Sprachen, wie diese Konzepte mit sprachspezifischen Designzielen in Einklang gebracht werden können. In diesem Abschnitt sollen zunächst kurz die interessanten Aspekte zusammengefasst werden; anschließend nenne ich einige Probleme, die sich durch Anwendung des Objektparadigmas ergeben.

4.1 Interessante Aspekte der OOP

Objektorientierte Programmierung fördert die Lösung von Problemen durch Zerlegung in Teilprobleme. In OOP ist Abstraktion "natürlich": Will man eine umfangreiche Anwendung entwerfen, die Eingaben verarbeiten und die Ergebnisse anschließend graphisch darstellen, so begünstigen die meisten objektorientierten Sprachen die Kapselung der einzelnen Aspekte des Programmes in Objekte oder Klassen, die auf eine scharf umrissene Funktionalität beschränkt sind - in diesem Fall z.B. Klassen für das Lesen und Formatieren von Eingaben, Klassen zum Verarbeiten der formatierten Eingabe und Klassen zur graphischen Präsentation der Ergebnisse.
An diesem Beispiel lässt sich ein weiterer häufig erwähnter Vorteil von OOP festmachen: Durch das Aufbauen von Objekt- und Klassenhierarchien erstellt man gleichzeitig Bibliotheken, die, wenn sorgfältig entwickelt, ein gewisses Maß an Wiederverwendbarkeit besitzen. Die Eingabeklassen z.B. kann man in allen Szenarien einsetzen, die Eingabeformatierung erfordern.

4.2 Probleme der OOP

Es existieren eine Reihe von Komplikationen, die sich im Zusammenhang mit OOP ergeben können. Um einige zu nennen:

Mehrfachvererbung In Sprachen wie C++ ist es einer Klasse möglich, von mehreren anderen Klassen zu erben. Dies führt zu Effekten, die Sprachentwickler vor schwierige Designetscheidungen stellen:

- Was geschieht, wenn zwei Methoden mit gleichem Typ von zwei Oberklassen geerbt werden müssen?
- Was geschieht, wenn eine Klasse D von zwei Klassen B, C erbt, die ihrerseits eine gemeinsame Oberklasse A haben? Die besondere Komplikation hierbei besteht darin, dass die Klassen "in der Mitte" unterschiedlichen Gebrauch vom Zustand (den Daten) der Klasse A machen können und sich folglich ihre Methoden gegenseitig durch Seiteneffekte beeinflussen können.

Um diesen Problemen aus dem Weg zu gehen, erlauben manche Sprachen (wie z.B. Java) nur Einzelvererbung. Um das Subtyping expressiver zu machen, wird allerdings das Erben von mehreren *Interfaces* erlaubt. Dabei handelt es sich im Grunde um abstrakte Klassen (siehe C++), die keine Implementierungen enthalten.

"Inheritance is not subtyping" Subklassen sind oft keine Subtypen. Definiert man zum Beispiel eine Klasse `Point2D` , deren Instanzen Punkte im zweidimensionalen Raum darstellen, zusammen mit einer Prozedur `equal : Point2D -> bool` , die zwei Punkte auf Gleichheit testet, so verliert man das echte Subtyping, wenn in einer abgeleiteten Klasse `Point3D` die Prozedur durch `equal : Point3D -> bool` ersetzt wird. Der Grund hierfür ist die im ersten Kapitel vorgestellte Kontravarianzregel.

Privilegierter Zugriff in binären Methoden Sobald man Prozeduren als zu einem Objekt gehörig betrachtet, räumt man diesem Objekt eine Vorangstellung ein: Die Implementierung des Objekts entscheidet über den auszuführenden Code. Ist nun ein Argument der Prozedur eine unterschiedliche Implementierung desselben Objekttyps (desselben Interfaces etc.), so können Probleme auftreten: Die Implementierung, deren Methode ausgeführt wird, hat keine Kenntnis über die Implementierung des Argumentobjekts und kann daher nur über die Interface-Methoden darauf zugreifen. Sind diese aber für eine bestimmte Operation nicht ausreichend, muss der Programmierer das Interface erweitern. Dies kann aber mit einer der beiden Implementierungen konfligieren, da neue Operationen vielleicht nur umständlich und unter Verlust von Optimierungen einführbar sind.

Verwendete Literatur

- John C. Mitchell: *Concepts in Programming Languages. Cambridge University Press 2003*
- Kim Bruce, Luca Cardelli, Giuseppe Castagna, Hopkins Object Group, Gary T. Leavens, Benjamin Pierce: *On Binary Methods. Theory And Practice of Object Systems*
- Richard P. Gabriel: *Objects Have Failed. OOPSLA Debate 2002*
- William Cook, Walter Hill, Peter Canning: *Inheritance Is Not Subtyping. Principles of Programming Languages, 1990*